Program Verification for Optimized Byte Copy

Edoardo S. Biagioni

July 1994
CMU-CS-94-172

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also published as Fox Memorandum CMU-CS-FOX-94-06

Abstract

We present a program to copy bytes, together with a formal specification and a proof that the code satisfies the specification. The program, which is in the critical path for a network implementation, has been tuned carefully over a period of time; the proof covers the entire program, and is easily updated if the program is modified. The program is written in the Standard ML programming language and was produced as part of the Fox Project implementation of the TCP/IP protocol suite.

The author's electronic mail address is: esb@cs.cmu.edu

This research was sponsored by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628-91-C-0168.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AD NUMBER	DATE	DTIC ACCESSION NOTICE
1. REPORT IDENTIFYING INFORMATION		**REQUESTER:** 1. Put your mailing address on reverse of form. 2. Complete items 1 and 2. 3. Attach form to reports mailed to DTIC. 4. Use unclassified information only. 5. Do not order document for 6 to 8 weeks.
A. ORIGINATING AGENCY CARNEGIE MELLON UNIV		
B. REPORT TITLE AND/OR NUMBER CMU-CS-94-172		
C. MONITOR REPORT NUMBER		
D. PREPARED UNDER CONTRACT NUMBER ARPA/CSTO F19628-91-C-0168		
2. DISTRIBUTION STATEMENT (A) UNLIMITED		**DTIC:** 1. Assign AD Number. 2. Return to requester.

DTIC Form 50 DEC 91 PREVIOUS EDITIONS ARE OBSOLETE

1 Introduction

Data copying is a common operation in systems code. Because it is frequently used, it is important that the implementation of data copying be optimized. Operating systems commonly provide a byte-copy library function that, for maximum performance, is written in assembly.

Assembly lets programmers optimize the performance of a specific algorithm, but higher-level languages let programmers easily try out different algorithms to discover which might be the most efficient. With sufficiently good compilers and sufficiently complicated architectures, a program in a more advanced language such as Standard ML (SML) or Lisp can even outperform one carefully written in a lower-level language [1], though for the copy code presented here we claim no such result[1]. One undesirable consequence of trying out different algorithms is the possibility of introducing errors. The main motivation for the proof presented in this paper was preventing the introduction of errors during optimization of a byte copy function written in SML.

The errors were discovered by Nevin Heintze when he applied set-based analysis [5] to the code and noticed that the reported ranges for some of the array indices were outside the legal ranges. After verifying that the error was not in the set-based analysis code, the errors were shown to be in the copy code.

The discovery of such errors made us aware that more errors might be present. The byte copy code is imperative in style, and optimizations often obscure the meaning of the program; the copy code is therefore more prone to errors than most SML programs. Because of this, and because the module is easily specified, the author decided that a proof of correctness would be an effective strategy for increasing confidence in the correctness of the program.

The first proof was written in the pre-condition and post-condition style similar to that first suggested by Naur [8] and Floyd [3]. The proof was successful in that the resulting program, though exhaustively tested, showed no implementation bugs[2]. The inadequacy of this first proof became apparent, however, when it came time to further optimize the code: the proof was unmaintainable.

Further work by the author, with advice from Matthias Felleisen and Robert Harper, produced a transformational proof similar to those described by Mason [6]. The program has been modified since the proof was completed, and the proof modified with it, at little more cost in programmer time than it would take to describe and motivate in detail all the changes.

The author believes the significance of this work to lie in its being a practical and maintainable transformational proof of an optimized program. The transformational style of program proofs was selected in part because it is easily modified when the program is modified, leading to a maintainable proof.

The proof and the program were developed together; the author believes this strategy leads to simpler proofs and clearer and better programs than developing the program and attempting to prove it, or developing a proof and attempting to implement it.

The remainder of this paper describes the proof and the source code in greater detail. Section 2 describes some related research, Sections 3–13 give the specification, the source code, and the proof of the program, and Sections 14 and 15 summarize the work and point to some directions for further research.

---

[1] However, our checksum code written in SML does outperform a production checksum code written in C which uses a less efficient algorithm.

[2] The exhaustive testing did bring to light one specification error. Once the specification was amended and the code and proof changed correspondingly, the program tested correctly the first time it was run.

2 Background

2.1 The Fox Project

The Fox Project at CMU [2] [4] is exploring the possibilities offered by an advanced programming language for implementing systems code. Currently the Fox Project is implementing networking protocols such as TCP and IP using the Standard ML (SML) [7] language. SML offers an elaborate module system, polymorphic static type checking, and a notion of safety that guarantees that operations on one data value will not inadvertently affect other data values. The latter is not true in C, for example: an operation on a pointer which was initialized to point to one data value may well affect other, unrelated data values. This kind of safety is useful in preventing bugs whose manifestation is usually far-removed from the cause, and are therefore hard to locate.

To maintain this kind of safety, as much as possible of the code is implemented in SML; this includes the byte copy code presented in this paper. As well as the safety property, characteristics of SML that are useful for proofs include the explicit identification of those objects whose value may change, a lack of explicit pointers, and the scoping introduced by modules.

At present (July 1994) we use a modified version of the SML/NJ compiler version 0.93; this compiler does not especially optimize loops over arrays. To obtain acceptable performance, we have performed by hand several classical optimizations such as loop unrolling, common-subexpression elimination, and case analysis. The success of this optimization can be seen by the fact that, on a typical system such as the DECstation 5000/120, the maximum performance for the highly optimized aligned copy, 3.4 megabytes per second, is over four times the 770 kilobytes per second of the unoptimized unaligned copy. This performance is similar to that of a simple C program that copies 4-byte words from one array to another, about 3.0 megabytes per second using the native C compiler (with `-O` optimization), though much slower than that of a simple C program that copies 4-byte words between arrays by using pointers, about 23 megabytes per second, or of the library function **bcopy**, about 33 megabytes per second.

2.2 Proof Techniques

There are many proof techniques that are useful for proving algorithms. Of these, some are potentially useful for proving programs. With traditional pre-conditions and post-conditions [8] [3] [9], a program is proven by producing assertions that hold at different points in a program, and verifying that the pre-conditions together with the code can be used to derive the postconditions. More recent work by Scherlis and Scott [10] introduces the idea that proofs should be maintainable; within this context Mason presents proof techniques suitable for building and maintaining proofs of destructive programs [6], that is, programs which are not purely functional. The technique presented by Mason is to prove that programs or program fragments are *strongly isomorphic* to one another, that is, they return the same value and have the same effect on the store; if two programs or program fragments are strongly isomorphic, then one can replace the other without affecting the meaning of the program. Though Lisp is the language used by Mason, the same techniques are applicable to SML programs. The proof presented in this paper uses all of these techniques.

The proof is divided into three parts. We first prove that the main function satisfies the specification, by showing that its specification is equivalent to that of its subsidiary functions and that exactly one of these functions is called by the main function.

We then prove that a *base case*, a simple, unoptimized routine, implements the specification. This proof uses traditional pre-condition and post-condition techniques and a counting argument equivalent to induction over the loop.

We finally transform the base case into the optimized routines using only simple, provably correct steps, with techniques similar to those described by Mason [6]. The correctness of the main function is derived from the specification of the other routines and the assumption that the routines are correct.

In all these proofs we assume that the program type checks, and that therefore there are no type errors. The proof itself is therefore largely typeless, and if fact loose in mixing 2-byte and 4-byte values.

2.3 Notation and Organization

The notation used in this paper is shown in Table 2.3. The code uses **andb** instead of the SML **mod** operation since SML/NJ 0.93 does not optimize modulo operations where the second operand is a constant power of two.

Symbol	Meaning
$\simeq$	program equivalence (strong isomorphism)
$\ll$	left shift
$\gg$	right shift
&	bit-wise logical *and*
|	bit-wise logical *or*
`andb(x, m-1)`	$x \bmod 2^m$

Table 1: Notation

The code is shown in the order in which it is defined in the original source, except that the definition of the main **copy** function is at the beginning instead of the very end.

Though knowledge of SML may be helpful, it should not be required for understanding this paper. The SML used in this program includes the Fox extensions to SML/NJ, specifically the **Byte1**, **Byte2**, and **Byte4** structures that provide operations on 8-bit, 16-bit, and 32-bit unsigned integers; those used in this code are array subscripting and update, bit shifts, logical *and*, and logical *or*.

3 Functor Header

```
functor Copy (structure TimingBoard: TIMINGBOARD): COPY =
 struct
   (* raised if the copy would access data outside one of the arrays *)
  exception Illegal_Copy of {source_length: int, source_offset: int,
                             bytes: int, dest_length: int, dest_offset: int}
  exception Illegal_Align_Value_In_Copy
        (* never raised; avoids compiler warnings *)
  val copy_counter = TimingBoard.add "copy"
```

4 Function copy

The following assumptions are used in the proof; the proof is only valid if the assumptions hold.

1. The computations cause no arithmetic overflow.

2. **TimingBoard.time** returns the result of applying its function to unit, so we can ignore **TimingBoard.time** when verifying correctness.

3. Any out-of-bounds array access will raise the exception **Subscript**.

4. *Byte4.endian* = *Byte4.Little* implies that *Byte2.update*$(a, 0, x)$; *Byte2.update*$(a, 2, y) \simeq$ *Byte4.update*$(a, 0, x|y \ll 16)$

5. *Byte4.endian* $\neq$ *Byte4.Little* implies that *Byte2.update*$(a, 0, x)$; *Byte2.update*$(a, 2, y) \simeq$ *Byte4.update*$(a, 0, x \ll 16|y)$

4.1 Preconditions

1. $src \neq dest$ (if $src = dest$, the meaning of the function is undefined).

2. $\forall j \in \{0 \ldots length(dest) - 1\} : dest_j = orig_j$.

4.2 Specification

1. $\forall i \in \{0 \ldots length(src) - 1\}$: the value of src_i is not changed by this call.

2. if $bytes \leq 0$ then **copy** $\simeq$ null function. After the call, $\forall j \in \{0 \ldots length(dest) - 1\} : dest_j = orig_j$.

3. otherwise if $srcpos < 0 \vee destpos < 0$ then **copy** $\simeq$ **raise Illegal_Copy**.

4. otherwise if $srcpos + bytes - 1 \geq length(src) \vee destpos + bytes - 1 \geq length(dest)$ then **copy** $\simeq$ **raise Illegal_Copy**.

5. otherwise, after the call $\forall j \in \{0 \ldots length(dest) - 1\}$: if $destpos \leq j < destpos + bytes$ then $dest_j = src_{j+srcpos-destpos}$, otherwise $dest_j = orig_j$

Note that when **Illegal_Copy** is raised, *dest* may or may not be modified.

4.3 Proof

The specification of **copy** can be mapped to that of **unaligned** by the following changes:

1. use *endsrc* in **unaligned** instead of *srcpos* + *bytes* in **copy**.

2. **unaligned** raises **Subscript** exactly when **copy** raises **Illegal_Copy**,

With these changes, **copy** satisfies its specification if it is equivalent to calling **unaligned**.

copy has no side effects except those produced by calling **common**, **aligned**, **semialigned**, or **unaligned**; in every case copy will call exactly one of these functions, with as *endsrc* argument the value of *srcpos* + *bytes*. Since all these functions satisfy the specification of **unaligned** if their preconditions are met, **copy** satisfies its specification if the preconditions of each function are met.

The preconditions of **unaligned** and **common** are the same as those of **copy**.

The preconditions of **aligned** are that *srcpos* mod 4 = *destpos* mod 4 = *srcalign* and *bytes* $\geq$ 4, and these are guaranteed by the negation of **bytes < 25** and the assertion of **srcalign = destalign**.

The preconditions of **semialigned** are that *srcpos* mod 4 = (*destpos* + 2) mod 4 = *srcalign* and *bytes* $\geq$ 12, and these are guaranteed by the negation of `bytes < 25` and of `srcalign = destalign`, which means that *srcpos* mod 4 $\neq$ *destpos* mod 4, and the assertion of `Bits.andb (srcalign + destalign, 0x1) = 0`, which means that *srcpos* mod 2 = *destpos* mod 2, so that we can conclude that *srcpos* mod 4 = *destpos* + 2 mod 4.

The **copy** function therefore satisfies the specification.

4.4 Code

```
fun copy (src, srcpos, bytes, dest, destpos) =
     (TimingBoard.time (copy_counter, fn () =>
     (if bytes < 25 then
(* cannot call aligned or semialigned with less than 12 bytes,
   but we go up to 25 to optimize common cases. *)
       common (src, srcpos, srcpos + bytes, dest, destpos)
      else
       let val srcalign = Bits.andb (srcpos, 0x3)
           val destalign = Bits.andb (destpos, 0x3)
           val endsrc = srcpos + bytes
       in if srcalign = destalign then
           aligned (src, srcpos, endsrc, dest, destpos, srcalign, bytes)
          else if Bits.andb (srcalign + destalign, 0x1) = 0 then
           semialigned (src, srcpos, endsrc, dest, destpos,
                        srcalign, bytes)
          else
           unaligned (src, srcpos, endsrc, dest, destpos)
       end)
        handle Subscript => illegal (src, srcpos, bytes, dest, destpos)))
end (* struct *)
```

5 Function illegal

5.1 Specification

`illegal` $\simeq$ `raise Illegal_Copy`

5.2 Proof

If we ignore the print statement, clearly `illegal` $\simeq$ `raise Illegal_Copy`. However, the print statement terminates and modifies no existing memory, so it does not change the semantics of **illegal**.

5.3 Code

```
fun illegal (src, srcpos, bytes, dest, destpos) =
    (print ("copy.fun: illegal copy (" ^
            makestring (ByteArray.length src) ^ ", " ^
            makestring srcpos ^ ", " ^
            makestring bytes ^ ", " ^
            makestring (ByteArray.length dest) ^ ", " ^
            makestring destpos ^ ")\n");
     raise Illegal_Copy {source_length = ByteArray.length src,
                         source_offset = srcpos,
                         bytes = bytes,
                         dest_length = ByteArray.length dest,
                         dest_offset = destpos})
```

6 Function unaligned

This is the basic function for copying bytes. We prove that this is correct, then transform it to the more specific, more optimized versions.

6.1 Preconditions

1. $src \neq dest$ (if $src = dest$, the meaning of the function is undefined).
2. $\forall j \in \{0 \ldots length(dest) - 1\} : dest_j = orig_j$.

6.2 Specification

1. $\forall i \in \{0 \ldots length(src) - 1\}$: the value of src_i is not changed by this call.
2. if $endsrc - srcpos \leq 0$ then copy $\simeq$ null function. After the call, $\forall j \in \{0 \ldots length(dest) - 1\}$: $dest_j = orig_j$.
3. otherwise if $srcpos < 0 \vee destpos < 0$ then copy $\simeq$ `raise Subscript`.
4. otherwise if $srcpos + endsrc - srcpos - 1 \geq length(src) \vee destpos + endsrc - srcpos - 1 \geq length(dest)$ then copy $\simeq$ `raise Subscript`.
5. otherwise, after the call $\forall j \in \{0 \ldots length(dest) - 1\}$: if $destpos \leq j < endsrc + destpos - srcpos$ then $dest_j = src_{j+srcpos-destpos}$, otherwise $dest_j = orig_j$

6.3 Proof

The sections of this proof correspond to those of the specification.

1. There is only one **update** operation in the code, and it updates *dest*; therefore, *src* is not modified.
2. if $endsrc - srcpos \leq 0$ then $srcpos \geq endsrc$. On the first call to `loop`, $i = srcpos$, so $i \geq endsrc$, the else clause is not entered, and the function terminates immediately. Therefore, **unaligned** $\simeq$ null function.

3. otherwise, if $srcpos < 0$ or $destpos < 0$ then on the first call to `loop`, when $i = srcpos$ and $j = destpos$ and since $i < endsrc$, we read src_i and update $dest_j$. Therefore if $srcpos < 0$ we raise `Subscript`, and likewise if $destpos < 0$.

4. otherwise, if $srcpos + endsrc - srcpos - 1 \geq length(src)$ then $endsrc - 1 \geq length(src)$. Since $srcpos < endsrc$, the else part of the loop is entered at least once when $i = endsrc - 1$, and the operation `Byte4.sub (src, i)` will then raise `Subscript`. Replacing $srcpos$ by $destpos$, i by j, and src by $dest$ proves that if $destpos + endsrc - srcpos - 1 \geq length(dest)$ then **unaligned** $\simeq$ `raiseSubscript`.

5. otherwise, we have that $srcpos \geq 0$, $destpos \geq 0$, $srcpos < endsrc \leq length(src)$, and $destpos < destpos + endsrc - srcpos \leq length(dest)$. Therefore, the else part of the loop is entered $endpos - srcpos - 1$ times, with i taking on all the values in the range $srcpos \ldots endpos - 1$ inclusive and j the values in the range $destpos \ldots destpos + endpos - srcpos - 1$ inclusive; i and j are related by $i = j + srcpos - destpos$.

   Therefore **unaligned** terminates, and when it terminates, $\forall j \in \{destpos \ldots destpos + endsrc - srcpos - 1\} : dest_j = src_{j+srcpos-destpos}$. The other locations in $dest$ are unchanged: $\forall j \in \{0 \ldots length(dest) - 1\} - \{destpos \ldots destpos + endsrc - srcpos - 1\} : dest_j = orig_j$.

6.4 Code

```
fun unaligned (src, srcpos, endsrc, dest, destpos) =
    let fun loop (i, j) =
            if i >= endsrc then ()
            else
             (Byte1.update (dest, j, Byte1.sub (src, i));
              loop (i + 1, j + 1))
    in loop (srcpos, destpos)
    end
```

7 Function common

This function optimizes the implementation of short copies of 1, 2, 3, 4, 8, 16, or 20 bytes.

The preconditions and specification are the same as for **unaligned**.

7.1 Proof

We can transform **unaligned** into this function by a sequence of steps, none of which changes the meaning of the function:

1. Rename i and j to $srcpos$ and $destpos$, and inline `loop`.

2. Break into cases, use a call to **unaligned** for the default case, and unroll the loop the appropriate number of times for the specific cases.

7.2 Code

```
fun common (src, srcpos, endsrc, dest, destpos) =
    case endsrc - srcpos of
      0 => ()
```

```
| 1 =>
   Byte1.update (dest, destpos, Byte1.sub (src, srcpos))
| 2 =>
   (Byte1.update (dest, destpos, Byte1.sub (src, srcpos));
    Byte1.update (dest, destpos + 1, Byte1.sub (src, srcpos + 1)))
| 3 =>
   (Byte1.update (dest, destpos, Byte1.sub (src, srcpos));
    Byte1.update (dest, destpos + 1, Byte1.sub (src, srcpos + 1));
    Byte1.update (dest, destpos + 2, Byte1.sub (src, srcpos + 2)))
| 4 =>
   (Byte1.update (dest, destpos, Byte1.sub (src, srcpos));
    Byte1.update (dest, destpos + 1, Byte1.sub (src, srcpos + 1));
    Byte1.update (dest, destpos + 2, Byte1.sub (src, srcpos + 2));
    Byte1.update (dest, destpos + 3, Byte1.sub (src, srcpos + 3)))
| 8 =>
   (Byte1.update (dest, destpos, Byte1.sub (src, srcpos));
    Byte1.update (dest, destpos + 1, Byte1.sub (src, srcpos + 1));
    ...
    Byte1.update (dest, destpos + 7, Byte1.sub (src, srcpos + 7)))
| 16 =>
   (Byte1.update (dest, destpos, Byte1.sub (src, srcpos));
    Byte1.update (dest, destpos + 1, Byte1.sub (src, srcpos + 1));
    ...
    Byte1.update (dest, destpos + 15, Byte1.sub (src, srcpos + 15)))
| 20 =>
   (Byte1.update (dest, destpos, Byte1.sub (src, srcpos));
    Byte1.update (dest, destpos + 1, Byte1.sub (src, srcpos + 1));
    ...
    Byte1.update (dest, destpos + 19, Byte1.sub (src, srcpos + 19)))
| _ => unaligned (src, srcpos, endsrc, dest, destpos)
```

8 Function sixteen

This function copies bytes when both the source and destination are aligned on four-byte boundaries.

8.1 Preconditions

1. $src \neq dest$ (same as for **unaligned**).
2. $\forall j \in \{0 \ldots length(dest) - 1\} : dest_j = orig_j$ (same as for **unaligned**).
3. $srcpos \bmod 4 = destpos \bmod 4 = 0$
4. $(endsrc - srcpos) \bmod 16 = 0$

8.2 Specification

Same as for **unaligned**.

8.3 Proof

We can transform **unaligned** into this function through the following steps. None of the steps changes the meaning of the function.

1. Unroll the loop in **unaligned** four times, to give

```
fun loop (i, j) =
     if i >= endsrc then ()
     else
      (Byte1.update (dest, j + 0, Byte1.sub (src, i + 0));
       Byte1.update (dest, j + 1, Byte1.sub (src, i + 1));
       Byte1.update (dest, j + 2, Byte1.sub (src, i + 2));
       Byte1.update (dest, j + 3, Byte1.sub (src, i + 3));
       loop (i + 4, j + 4))
```

   The unrolled loop is strongly isomorphic to the original, since $(endsrc - srcpos) \bmod 16 = 0$. so $(endsrc - srcpos) \bmod 4 = 0$.

2. Merge the four calls to **Byte1.sub** and the four calls to **Byte1.update** into single calls to **Byte4.sub** and **Byte4.update** respectively.

```
fun loop (i, j) =
     if i >= endsrc then ()
     else
      (Byte4.update (dest, j + 0, Byte4.sub (src, i + 0));
       loop (i + 4, j + 4))
```

   Since *srcpos* mod 4 = *destpos* mod 4 = 0, this substitution preserves the meaning of the previous loop.

3. Unroll the loop again four times to give the final code. The unrolled loop is strongly isomorphic to the original since $(endsrc - srcpos) \bmod 16 = 0$.

8.4 Code

```
local
 fun sixteen (src, srcpos, endsrc, dest, destpos) =
      let fun loop (i, j) =
               if i >= endsrc then ()
               else
                (Byte4.update (dest, j,      Byte4.sub (src, i));
                 Byte4.update (dest, j + 4,  Byte4.sub (src, i + 4));
                 Byte4.update (dest, j + 8,  Byte4.sub (src, i + 8));
                 Byte4.update (dest, j + 12, Byte4.sub (src, i + 12));
                 loop (i + 16, j + 16))
      in loop (srcpos, destpos)
      end
```

9 Function aligned

9.1 Preconditions

1. $src \neq dest$ (same as for **unaligned**).
2. $\forall j \in \{0 \ldots length(dest) - 1\} : dest_j = orig_j$ (same as for **unaligned**).
3. $srcpos \bmod 4 = destpos \bmod 4 = srcalign$
4. $srcpos + bytes = endsrc$
5. $bytes \geq 4$

9.2 Specification

Same as for **unaligned**.

9.3 Proof

We can transform **unaligned** into this function through the following steps. None of the steps changes the meaning of the function.

1. Add the parameters *srcalign* and *bytes* and the **let**:

```
fun aligned (src, srcpos, endsrc, dest, destpos, srcalign, bytes) =
    let val front = case srcalign of
                      0 => 0
                    | 1 => 3
                    | 2 => 2
                    | 3 => 1
        ...
        val backdest = middest + middle
    in unaligned (src, srcpos, endsrc, dest, destpos)
    end (* let *)
```

   This is equivalent to **unaligned** since the **let** only introduces pure, finite computations. Since $0 \leq srcalign = srcpos \bmod 4 < 4$, the exception `Illegal_Align_Value_In_Copy` is never raised.

2. Split the call to **unaligned** into:

```
unaligned (src, srcpos, midsrc, dest, destpos);
sixteen (src, midsrc, backsrc, dest, middest);
unaligned (src, backsrc, endsrc, dest, backdest)
```

   This is equivalent to the original call, since the range $srcpos \ldots midsrc \ldots backsrc \ldots endsrc$ is the same as the range $srcpos \ldots endsrc$, and likewise for *dest*, provided the preconditions of **sixteen** are maintained:

(a) $midsrc \bmod 4 = 0$ follows from precondition 3 and from $midsrc \bmod 4 = (srcpos + (4 - srcalign)) \bmod 4 = (srcpos + (4 - srcpos \bmod 4)) \bmod 4 = (4 + srcpos - srcpos) \bmod 4 = 0$. Likewise $middest \bmod 4 = 0$, since $middest = destpos + front$ as $midsrc = srcpos + front$ and $srcpos \bmod 4 = destpos \bmod 4$.

(b) $(backsrc - midsrc) \bmod 16 = 0$ follows from $backsrc - midsrc = middle = rest - (rest \bmod 16)$ and $rest \geq 0$ since $rest = bytes - x$ for some $x < 4$ and $bytes \geq 4$.

9.4 Code

```
in (* local *)
 fun aligned (src, srcpos, endsrc, dest, destpos, srcalign, bytes) =
      let val front = case srcalign of
                        0 => 0
                      | 1 => 3
                      | 2 => 2
                      | 3 => 1
                      | _ => raise Illegal_Align_Value_In_Copy
          val rest = bytes - front
          val tail = Bits.andb (rest, 0xf)
          val middle = rest - tail
          val midsrc = srcpos + front
          val middest = destpos + front
          val backsrc = midsrc + middle
          val backdest = middest + middle
      in unaligned (src, srcpos, midsrc, dest, destpos);
         sixteen (src, midsrc, backsrc, dest, middest);
         unaligned (src, backsrc, endsrc, dest, backdest)
      end (* let *)
end (* local *)
```

10 Function eightlittle

10.1 Preconditions

1. $src \neq dest$ (same as for **unaligned**).
2. $\forall j \in \{0 \ldots length(dest) - 1\} : dest_j = orig_j$ (same as for **unaligned**).
3. $srcpos \bmod 4 = (destpos + 2) \bmod 4 = 2$
4. $endsrc > srcpos$
5. $(endsrc - srcpos) \bmod 8 = 2$
6. $\texttt{Byte2.update}(a, 0, x); \texttt{Byte2.update}(a, 2, y) \simeq \texttt{Byte4.update}(a, 0, (x | y \ll 16))$

10.2 Specification

Same as for **unaligned**.

10.3 Proof

We can transform **unaligned** into this function through the following steps. None of the steps changes the meaning of the function.

1. Unroll the loop eight times, and since (*endsrc* − *srcpos*) mod 8 = 2, at the beginning copy the two bytes that cannot be copied by the loop. Then merge each pair of `Byte1.sub` and of `Byte1.update` operations into the corresponding `Byte2` operations. We can do this since *srcpos* mod 2 = *i* mod 2 = *destpos* mod 2 = *j* mod 2 = 0.

```
let fun loop (i, j) =
        if i >= endsrc then ()
        else (Byte2.update (dest, j + 0, Byte2.sub (src, i + 0));
              Byte2.update (dest, j + 2, Byte2.sub (src, i + 2));
              Byte2.update (dest, j + 4, Byte2.sub (src, i + 4));
              Byte2.update (dest, j + 6, Byte2.sub (src, i + 6));
              loop (i + 8, j + 8))
 in Byte2.update (dest, destpos. Byte2.sub (src, srcpos));
    loop (srcpos + 2, destpos + 2)
 end
```

2. Introduce a carry, and within the body of the loop add two to *i*, so now *j* = *i* + *destpos* − *srcpos* − 2. To keep equivalence with **unaligned**, read the first two bytes before the loop and assign the final carry at the end of the loop. In this loop, *i* mod 4 = (*srcpos* + 2) mod 4 = 0 and *j* mod 4 = *destpos* mod 4 = 0.

```
let fun loop (i, j, carry) =
        if i >= endsrc then Byte2.update (dest, j, carry)
        else
         (Byte2.update (dest, j + 0, carry);
          Byte2.update (dest, j + 2, Byte2.sub (src, i + 0));
          Byte2.update (dest, j + 4, Byte2.sub (src, i + 2));
          Byte2.update (dest, j + 6, Byte2.sub (src, i + 4));
          loop (i + 8, j + 8, Byte2.sub (src, i + 6)))
in loop (srcpos + 2, destpos, Byte2.sub (src, srcpos)))
```

3. Replace the first two `Byte2.update` and `Byte2.sub` operations by corresponding `Byte4` operations, since *i* mod 4 = *j* mod 4 = 0. Use precondition 6 to compute the value to be stored.

```
val makebyte2 = Byte2.from_int o Byte4.to_int
val makebyte4 = Byte4.from_int o Byte2.to_int
fun mergel (low, high) = Byte4.|| (low, Byte4.<< (high, 16))
       ...
        else
         let val srcv = Byte4.sub (src, i)
         in Byte4.update (dest, j, mergel (carry, srcv));
            let val carry = makebyte2 (Byte4.>> (srcv, 16))
            in Byte2.update (dest, j + 4, carry);
               Byte2.update (dest, j + 6, Byte2.sub (src, i));
               loop (i + 8, j + 8,
                     makebyte4 (Byte2.sub (src, i + 6)))
            end
         end
       ...
```

4. Replace the remaining Byte2 operations by the corresponding Byte4 operations.

```
let val srcv = Byte4.sub (src, i)
in Byte4.update (dest, j, mergel (carry, srcv));
   let val carry = Byte4.>> (srcv, 16)
       val srcv = Byte4.sub (src, i + 4)
   in Byte4.update (dest, j + 4, mergel (carry, srcv));
      loop (i + 8, j + 8, Byte4.>> (srcv, 16))
   end
end
```

5. In-line the calls to mergel and compute new i, j as $i + 4, j + 4$.

```
...
let val srcv = Byte4.sub (src, i)
in Byte4.update (dest, j,
                 Byte4.|| (carry,
                           Byte4.<< (srcv, 16)));
   let val i = i + 4
       val j = j + 4
       val carry = Byte4.>> (srcv, 16)
       val srcv = Byte4.sub (src, i)
   in Byte4.update (dest, j,
                    Byte4.|| (carry,
                              Byte4.<< (srcv, 16)));
      loop (i + 4, j + 4, Byte4.>> (srcv, 16))
...
```

This gives us the final code for **eightlittle**.

10.4 Code

```
local
 val makebyte2 = Byte2.from_int o Byte4.to_int
 val makebyte4 = Byte4.from_int o Byte2.to_int
 fun eightlittle (src, srcpos, endsrc, dest, destpos) =
      let fun loop (i, j, carry) =
              if i >= endsrc then Byte2.update (dest, j,
                                                makebyte2 carry)
              else
               let val srcv = Byte4.sub (src, i)
               in Byte4.update (dest, j,
                                Byte4.|| (carry, Byte4.<< (srcv, 16)));
                  let val i = i + 4
                      val j = j + 4
                      val carry = Byte4.>> (srcv, 16)
                      val srcv = Byte4.sub (src, i)
                  in Byte4.update (dest, j,
                                   Byte4.|| (carry,
                                             Byte4.<< (srcv, 16)));
                     loop (i + 4, j + 4, Byte4.>> (srcv, 16))
                  end
               end
      in loop (srcpos + 2, destpos,
              makebyte4 (Byte2.sub (src, srcpos)))
      end
```

11 Function eightbig

11.1 Preconditions

1. $src \neq dest$ (same as for **unaligned**).
2. $\forall j \in \{0 \ldots length(dest) - 1\} : dest_j = orig_j$ (same as for **unaligned**).
3. $srcpos \bmod 4 = (destpos + 2) \bmod 4 = 2$
4. $endsrc > srcpos$
5. $(endsrc - srcpos) \bmod 8 = 2$
6. $\texttt{Byte2.update}(a, 0, x); \texttt{Byte2.update}(a, 2, y) \simeq \texttt{Byte4.update}(a, 0, (x \ll 16 | y))$

11.2 Specification

Same as for **unaligned**.

11.3 Proof

The preconditions are the same as for **eightlittle**, except for precondition 6 which is reversed. This reversal is expressed in the code by the interchange of $\ll$ and $\gg$ and by the initial and final

shift operations that handle storing the two bytes of carry in the most significant half of the carry variable.

11.4 Code

```
fun eightbig (src, srcpos, endsrc, dest, destpos) =
    let fun loop (i, j, carry) =
            if i >= endsrc then
             Byte2.update (dest, j, makebyte2 (Byte4.>> (carry, 16)))
            else
             let val srcv = Byte4.sub (src, i)
             in Byte4.update (dest, j,
                              Byte4.|| (carry, Byte4.>> (srcv, 16)));
                let val i = i + 4
                    val j = j + 4
                    val carry = Byte4.<< (srcv, 16)
                    val srcv = Byte4.sub (src, i)
                 in Byte4.update (dest, j,
                                  Byte4.|| (carry,
                                            Byte4.>> (srcv, 16)));
                    loop (i + 4, j + 4, Byte4.<< (srcv, 16))
                end
             end
    in loop (srcpos + 2, destpos,
            Byte4.<< (makebyte4 (Byte2.sub (src, srcpos)), 16))
    end
```

12 Function eight

12.1 Preconditions

1. $src \neq dest$ (same as for **unaligned**).
2. $\forall j \in \{0 \ldots length(dest) - 1\} : dest_j = orig_j$ (same as for **unaligned**).
3. $srcpos \bmod 4 = (destpos + 2) \bmod 4 = 2$
4. $endsrc > srcpos$
5. $(endsrc - srcpos) \bmod 8 = 2$

12.2 Specification

Same as for **unaligned**.

12.3 Proof

The function **eight** is **eightlittle** if **Byte4.endian** = Byte4.little, and **eightbig** otherwise. Both functions implement the full specification, but they have different preconditions.

If `Byte4.endian` = Byte4.little, Assumption 4 in Section 4 applies and precondition 6 of **eightlittle** is satisfied, so we use **eightlittle**.

Otherwise, Assumption 5 in Section 4 applies and precondition 6 of **eightbig** is satisfied, so we use **eightbig**.

12.4 Code

```
val eight = if Byte4.endian = Byte4.Little then eightlittle
            else eightbig
```

13 Function semialigned

13.1 Preconditions

1. $src \neq dest$ (same as for **unaligned**).
2. $\forall j \in \{0 \ldots length(dest) - 1\} : dest_j = orig_j$ (same as for **unaligned**).
3. $srcalign = srcpos \bmod 4 = (destpos + 2) \bmod 4$
4. $endsrc - srcpos = bytes$
5. $bytes \geq 12$

13.2 Specification

Same as for **unaligned**.

13.3 Proof

We can transform **unaligned** into this function through the following steps. None of the steps changes the meaning of the function.

1. Add the parameters *srcalign* and *bytes* and the `let`:

```
fun semialigned (src, srcpos, endsrc, dest, destpos,
                 srcalign, bytes) =
    let val front = case srcalign of
                      0 => 2
                    | 2 => 0
                    | 1 => 3
                    | 3 => 1
        ...
        val backdest = middest + middle
    in unaligned (src, srcpos, endsrc, dest, destpos)
    end (* let *)
```

   This is equivalent to **unaligned** since the `let` only introduces pure, finite computations.

2. Split the call to unaligned into:

```
unaligned (src, srcpos, midsrc, dest, destpos);
eight (src, midsrc, backsrc, dest, middest);
unaligned (src, backsrc, endsrc, dest, backdest)
```

This is equivalent to the original call (since the range $srcpos \ldots midsrc \ldots backsrc \ldots endsrc$ is the same as the range $srcpos \ldots endsrc$, and likewise for *dest*), provided the preconditions of **eight** are maintained:

(a) $midsrc \bmod 4 = 2$ follows from precondition 3 and from $midsrc \bmod 4 = (srcpos + front) \bmod 4 = (srcpos + (6 - srcalign)) \bmod 4 = (srcpos + (6 - srcpos \bmod 4)) \bmod 4 = 2$. Likewise $(middest + 2) \bmod 4 = 2$, since $middest = destpos + front$ as $midsrc = srcpos + front$ and $srcpos \bmod 4 = (destpos + 2) \bmod 4$.

(b) $backsrc > midsrc$ follows from $backsrc - midsrc = middle$ and $front + middle + tail = bytes \geq 12$ (by precondition 5), and since $front \leq 3$ and $tail \leq 7$, $middle \geq 2$.

(c) $(backsrc - midsrc) \bmod 8 = 2$ follows from $backsrc - midsrc = middle = rest - ((rest - 2) \bmod 8))$ and $rest \geq 2$ since $rest \geq middle \geq 2$.

13.4 Code

```
in (* local *)
 fun semialigned (src, srcpos, endsrc, dest, destpos, srcalign, bytes) =
      let val front = case srcalign of
                        0 => 2
                      | 2 => 0
                      | 1 => 1
                      | 3 => 3
                      | _ => raise Illegal_Align_Value_In_Copy
          val rest = bytes - front
          val tail = Bits.andb (rest - 2, 0x7)
          val middle = rest - tail
          val midsrc = srcpos + front
          val middest = destpos + front
          val backsrc = midsrc + middle
          val backdest = middest + middle
      in unaligned (src, srcpos, midsrc, dest, destpos);
         eight (src, midsrc, backsrc, dest, middest);
         unaligned (src, backsrc, endsrc, dest, backdest)
      end (* let *)
end (* local *)
```

14 Summary

This paper describes a proof of an optimized, imperative function. On the negative side, the proof is informal in not stating all the axioms, not always referring to the axioms, and having neither lemmas no theorems. The entire program is only about 200 lines in length. The performance of the implementation is about as good as can be obtained using arrays and bounds checks, but slower than an equivalent C implementation using pointers. The proof was done entirely by hand. Finally,

the copy function is relatively easy to specify, so developing the formal specification of the intended semantics of the function was not an obstacle.

On the positive side, we note that unlike much of the published results about program proofs, we have actually designed a specification for and proven a substantial function in a running system, including all the I/O, exceptions, and optimizations one finds in a real-world program. In spite of its brevity, the program is somewhat complex. To see this, note that the proof for the function **unaligned** is a little more than a page in length. The proof of correctness for the common case is even simpler, much of the space being taken up with the verification of correct behavior under exceptional conditions.[3] In contrast the proof of the function **semialigned** and of its subroutines takes up several pages without much space being devoted to exceptional cases; a comparison of the source code for the routines is sufficient to understand why. In the function **eightlittle**, for example, the loop carries around an accumulator, shifts half-words around and uses them to build words, and in short we have not just an imperative program, but an imperative program that relies on the size of storage units and on the bit representation of values. Because of this complexity, we find the fact that the proof techniques are suitable for creating maintainable proofs to be a strong validation of the proof techniques we used.

It should be noted that in an environment where code is actively maintained, there is no certainty of correctness of a program, even with a proof, unless the proof is mechanically checked to correspond to and verify the source code. Even though the proof presented here may prove the correctness of this version of the program, there is no guarantee that future modifications of the program will be accompanied by correct changes in the proof. The effort of doing a program proof does however help uncover many implementation errors, so a proof does increase confidence in the correctness of code.

Many of the results of this work apply to many different proofs and programs. In particular, we claim that the transformational program proofs shown in this paper can be more easily maintained than pre-condition, post-condition style proofs or other conventional techniques. We also claim that developing the program and the proof together is the best way to develop simple proofs for clear and effective programs.

15 Future Work

We can speculate as to why we found it necessary to prove this particular function and not others. The motivations for the proof were twofold: the program was being changed frequently, and the program frequently had undetected bugs that were far from apparent on inspection. The frequent changes were due to our recognizing that this function was a bottleneck in our implementation of network protocols and to our attempts to optimize it; the mistakes may in part be due to the fact that, although a simple program, this is an imperative program with loops over arrays and word operations, and in SML as in other languages, loops over arrays and word operations are often error-prone.

Discussion with Matthias Felleisen made it apparent that the *array*, *position* pairs of parameters to the copy functions are really a form of a dynamic array where the first *position* bytes of the conventional, underlying array are never used. Perhaps if SML or other functional languages would provide arrays with the analogue of the "tail" and pattern matching operations provided for lists, the copy code might be simpler, potentially more efficient (the compiler could optimize the "tail"

[3]The function is sufficiently simple that the explicit proof could be omitted with little loss of confidence in the correctness of the program.

operation to pointer operations, avoiding the indexing required of arrays), and easier to understand and prove correct. Such "shrinkable" dynamic arrays might provide all of the performance benefits of pointers with none of the safety and obscurity problems pointers introduce. The program might even become sufficiently easy to understand that a correctness proof would no longer be required; this would be a desirable result.

16 Acknowledgements

None of this work would have been possible without the fruitful discussions the author had with Matthias Felleisen; the knowledge and inspiration he provided were both essential to the final result. Robert Harper and Peter Lee provided encouragement, motivation, and hope that the compiler would improve. Nevin Heintze was instrumental in causing the author to do a proof in the first place. Other members of the Fox project, including but not limited to Kenneth Cline, Mark Leone, Brian Milnes, Gregory Morrisett, and David Tarditi have all at one time or another provided useful feedback that led to the present form of the copy code.

References

[1] James M. Boyle and Terence J. Harmer. A practical functional program for the CRAY X-MP. *Journal of Functional Programming*, 2(1), 1992.

[2] Eric Cooper, Robert Harper, and Peter Lee. The Fox Project: Advanced development of systems software. Technical Report CMU-CS-91-178, School of Computer Science, Carnegie Mellon University, August 1991.

[3] R. W. Floyd. *Assigning Meanings to Programs*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[4] Robert Harper and Peter Lee. Advanced languages for systems software: The Fox project in 1994. Technical Report CMU-CS-94-01, School of Computer Science, Carnegie Mellon University, January 1991.

[5] Nevin Heintze. Set-based analysis of ML programs. In *1994 ACM Conference on Lisp and Functional Programming*, Orlando, Florida, June 1994.

[6] Ian Mason. *The Semantics of Destructive Lisp.* PhD thesis, Stanford University, 1986. also published in 1987 by the University of Chicago Press, CSLI Lecture Notes n. 5.

[7] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[8] P. Naur. Proof of algorithms by generated snapshots. *BIT*, 6(4), 1966.

[9] John C. Reynolds. *The Craft of Programming.* International Series in Computer Science. Prentice Hall, 1981.

[10] W. L. Scherlis and D. S. Scott. First steps towards inferential programming. In R. E. A. Mason, editor, *Information Processing 83*, New York, 1983. North Holland.